DSA Tutorial Week 5 - Graphs

Problem 0 - Preparation

You are given the following description of an undirected graph:

```
Vertices:   A, B, C, D, E
Edges:      AB, AC, BD, CD, DE
```

a. Represent this graph using an *adjacency matrix*, an *adjacency list*, and an *edge list*. Illustrate them using pen and paper.
b. After drawing/writing the graph in all three forms, analyze the space complexity of each representation and reflect on which type of graph (sparse or dense) is best suited for each representation.
c. Consider how different operations such as adding or removing edges, and finding adjacent vertices, are affected by the choice of representation.
d. To deepen your understanding and to verify your solution, use visualgo (https://visualgo.net/en/graphds) to visualize the graph in each representation form and compare it with your implementation.

Problem 1 - Finding Friends

a. You are given a set of persons **P** and their friendship relation **R**. That is, **(a, b)** ∊ **R** if and only if **a** is a friend of **b**. You must find a way to introduce person **x** to person **y** through a chain of friends.

   Model this problem with a graph and describe an algorithm to solve the problem.

   Pointers

   - First, think of a method that you know which solves the underlying, more general problem. If it is a shortest-path problem, should we traverse the vertices in a certain way?
   - To find the more general problem, think of what we are trying to achieve with the graph and try to express it in terms of collections of vertices and edges.
   - Now think of which adjustments are needed to the method to solve this specific instance of the more general problem.

b. If you were to implement this algorithm, which graph implementation would you choose for this problem? Why?

   Pointers

   - Determine how this problem can be modeled as a search for a path in a graph.
   - Consider using a graph traversal algorithm like Breadth-First Search (BFS) or Depth-First Search (DFS). Which would be most appropriate for this problem? What will that mean for your choice?

c. Implement the algorithm in java. Use ***FriendsChainFinder.java*** as a starting point. A solution is provided in the "answers" directory.

Reflection

- Reflect on how modeling real-world problems as graphs can provide clarity and a structured approach to problem-solving.

- Consider the modifications made to the standard algorithm to track the path. How did these changes enable the algorithm to solve the specific problem at hand?

- Reflect on the choice of an adjacency list for graph representation. How does this choice impact the algorithm's performance, especially in terms of space and time complexity?

TEACHER POINTERS

- Guide students to model persons as vertices and friendships as edges in an undirected graph. Discuss why an undirected graph is suitable due to the mutual nature of friendships.
- Emphasize **choosing the right data structure** for graph representation (like an adjacency list) and justify why it's efficient for sparse graphs and BFS traversal. *Students may not know what a sparse graph is so it may help to define it for them.*
- Explain the choice of BFS over other traversal methods. BFS is appropriate for finding the shortest path in unweighted graphs, which aligns with finding the shortest friendship chain.
- Discuss how BFS explores neighbors level by level, which ensures the shortest path is found.
- Focus on how to keep track of the path in BFS. This is not standard in a typical BFS implementation, so understanding how to modify it to store paths is crucial.
- Highlight the importance of checking if a vertex (friend) has already been visited to avoid cycles and redundant checks.
- **Encourage students to break down the problem**: first model the graph, then implement the BFS algorithm, and finally modify it to track and return the path.

Applying BFS in Different Implementations:

- **In Adjacency List:** BFS can efficiently traverse the graph by quickly accessing the list of friends for each person.
  An adjacency list is generally preferred for this problem, as it's likely more space-efficient and provides faster access to a person's friends, which is crucial for BFS traversal.
- **In Adjacency Matrix:** BFS would require checking each vertex to see if it's a friend, which can be less efficient, especially if the matrix is large and the graph is sparse.
  For a problem involving a social network, where not every person is friends with every other person, an adjacency matrix might not be the most space-efficient choice, especially for large networks.
- **In Edge List:** BFS would be inefficient as finding a vertex's neighbors (friends) requires scanning through all edges.
  Finding the neighbors of a particular vertex can be slow, as it might require scanning the entire list. This makes it less suitable for BFS in sparse graphs.

It may help to give an example of the different implementations.

Example Friendship Relations (draw this graph):
- Person 1 is friends with 2 and 4.
- Person 2 is friends with 1 and 3.
- Person 3 is friends with 2.
- Person 4 is friends with 1.

Adjacency List
```
1: [2, 4]
2: [1, 3]
3: [2]
4: [1]
```

Adjacency Matrix
```
   1  2  3  4
1 [0, 1, 0, 1]
2 [1, 0, 1, 0]
3 [0, 1, 0, 0]
4 [1, 0, 0, 0]
```

Edge List
```
[(1, 2), (1, 4), (2, 1), (2, 3), (3, 2), (4, 1)]
```

EXAMPLE SOLUTION

Graph Type:
The graph can be undirected since friendship is typically mutual. Use an Adjacency List because we are working on an algorithm that follows a path through the graph. Which means that the operation `get_neighbours` is the most often used.
Moreover, the number of friendships (edges) is likely much less than the square of the number of persons (vertices). An adjacency list is more space-efficient for sparse graphs.

Algorithm to Find a Chain of Friends:
Breadth-First Search (BFS):
- Use BFS starting from person x to find the shortest path to person y.
- BFS is ideal as it finds the shortest path in terms of the number of edges, which in this case represents the smallest number of intermediate friends.

Algorithm Steps:
- Create a queue and enqueue the starting person x along with a path list containing x.
- While the queue is not empty:
    - Dequeue a person and their current path.
    - For each friend of this person:
        - If the friend is y, return the path including this friend.
        - Else, enqueue the friend with the updated path.
- If y is not found, return that there is no chain of friends connecting x and y.

Example Pseudocode

```
function findFriendshipChain(P, R, x, y):
    Graph = buildGraph(P, R) // Build the graph from persons and relations
    Queue = initializeQueue()
    Queue.enqueue([x]) // Enqueue the starting person with the initial path

    while Queue is not empty:
        currentPath = Queue.dequeue()
        currentPerson = currentPath[-1] // Get the last person in the path

        if currentPerson == y:
            return currentPath

        for friend in Graph.getNeighbors(currentPerson):
            if friend not in currentPath: // Check to avoid revisiting
                newPath = currentPath + [friend]
                Queue.enqueue(newPath)

    return "No chain found" // Return if no path exists from x to y
```

Problem 2 - Difficult Parties

Consider a graph **G = (V, E)** which represents a social network in which each vertex represents a person, and an edge **(u, v)** ∊ **E** represents the fact that **u** and **v** know each other.

Your problem is to organize the largest party in which **nobody knows each other**. This is also called the *maximal independent set* problem.

Formally, given a graph **G = (V, E)**, find a set of vertices S of maximal size in which no two vertices are adjacent. (I.e., for all **u** ∊ **S** and **v** ∊ **S, (u, v)** ∉ **E**)

a. Define an algorithm for the Maximal Independent Set problem above. Write the algorithm in pseudocode.

   Pointers:

   - Begin by clarifying the definition of an independent set in a graph: a set of vertices where no two vertices are connected by an edge. The goal is to find the largest such set, known as the maximal independent set.
   - Consider an approach to identify independent sets. *For instance*, iterate through vertices and decide whether to include each vertex in the independent set based on its connections.
   - Implement a method to check if adding a vertex to the current set maintains the independence of the set.
   - Develop a strategy for incrementally constructing the independent set. This could involve greedy methods, backtracking, or other approaches, depending on the complexity you want to introduce.

b. Write a verification algorithm in pseudocode for the maximal independent set problem. This algorithm, called `testIndependentSet(G, S)`, takes a graph **G** represented through its adjacency matrix, and a set **S** of vertices, and returns true if **S** is a valid independent set for **G**.

c. Reflect on the time complexity of your Maximal Independent Set algorithm. What factors contribute to its complexity? Consider the process of iterating through vertices, checking for edges, and constructing the independent set. What do you think happens when you run your algorithm on a large graph?

d. Analyze the time complexity of the `testIndependentSet` function. What are the key operations in this function, and how do they contribute to its overall complexity? Does the size of the graph have a large impact on this algorithm? Does this result surprise you, when compared to the reflection in c.?

TEACHER POINTERS

- Discuss different graph representations (adjacency list, matrix, etc.) and their suitability for this problem.
- Guide students through the brute-force approach for finding maximal independent sets. Discuss why this approach is not feasible for large graphs due to its exponential time complexity.

- Help students understand the concept of an independent set in a graph and what makes the problem NP-hard. Explain that while solutions are hard to find, they are easy to verify.
- Emphasize the importance of the `testIndependentSet` function. Discuss how this function demonstrates the characteristic of NP problems: solutions are verifiable in polynomial time.

EXAMPLE SOLUTION

The Maximal Independent Set problem is an NP-hard problem, which means there is no known polynomial-time (*$O(n^k)$*) or lower) solution for it. However, we can use a brute-force approach to find a solution for small to moderately-sized graphs.

Here's an algorithm that uses a hashset of hashsets to find all independent sets and determine the largest:

```
Function findMaxSets(indSets):
    Initialize maxSets as Set of Sets
    Initialize maxCount as 0

    For each set in indSets:
        If size of set > maxCount:
            maxCount = size of set

    For each set in indSets:
        If size of set equals maxCount:
            Add set to maxSets

    Return maxSets

Function isSafe(vertex, solSet, edges):
    For each node in solSet:
        If edge exists between node and vertex in edges:
            Return false
    Return true

Function findAllSets(currV, setSize, solSet, verts, edges):
    Initialize allSets as Set of Sets

    For i from currV to setSize:
        If isSafe(verts[i - 1], solSet, edges):
            Add verts[i - 1] to solSet
            Call findAllSets(i + 1, setSize, solSet, verts, edges)
            Remove verts[i - 1] from solSet

    Add copy of solSet to allSets
    Return allSets
```

- **findMaxSets Function**: This function identifies the sets with the maximum number of vertices from a collection of independent sets (`indSets`).

    - It iterates through each set in `indSets`, tracking the size of the largest set found (`maxCount`).
    - It then adds all sets of this maximum size to a new collection (`maxSets`) and returns it.
- **isSafe Function**: This function checks if adding a vertex to the current solution set (`solSet`) will maintain its independence.
    - It iterates through each node in `solSet` and checks if an edge exists between this node and the new vertex. If an edge exists, the vertex cannot be added safely.
- **findAllSets Function**: This function generates all possible independent sets of vertices by recursively exploring combinations.
    - It iterates through the vertices (`verts`), adding a vertex to the `solSet` if it can be added safely (checked by `isSafe`).
    - It then recursively calls itself to continue building the set with the next vertex.
    - After exploring a branch (i.e., adding a vertex), it backtracks by removing the vertex and exploring other combinations.
    - Each independent set formed is added to `allSets`.

Complexity Analysis of the Algorithm:

1. findAllSets Function:

- The complexity of `findAllSets` is significant due to its recursive nature.
- For each vertex, the function makes a recursive call. In the worst case, this leads to exploring all combinations of vertices, which is exponential in terms of the number of vertices (O(2^n)).
- The backtracking approach ensures that all possible independent sets are considered, but at a high computational cost.

2. isSafe Function:

- The `isSafe` function checks for edges in `solSet`, which in the worst case requires checking against all other vertices. This gives it a complexity of O(n) per call.
- However, this is called within the recursive `findAllSets` function, contributing to its overall exponential complexity.

3. findMaxSets Function:

- This function iterates through each set in `indSets` twice, leading to a complexity of O(n), where n is the number of sets in `indSets`.
- Its complexity is overshadowed by the complexity of generating `indSets` in the first place.

Conclusion:

- The overall complexity of solving the Maximal Independent Set problem with this approach is dominated by the `findAllSets` function, which is exponential ***$O(2^n)$***.
- While the algorithm effectively identifies all possible independent sets and the maximal ones, its practical use is limited to graphs with a small number of vertices due to its high computational cost.

This algorithm tries to find all pairs of non-adjacent vertices and adds them to a set, storing these sets in a collection. It then finds the largest set from this collection.

```
function testIndependentSet(G, S):
    for each vertex u in S:
        for each vertex v in S:
            if u ≠ v and there is an edge between u and v in G:
                return false
    return true
```

This verification algorithm checks if any two vertices in set S are adjacent in graph G. If any adjacent pair is found, it returns false, indicating that S is not an independent set. If no adjacent pairs are found, it returns true. Because this is an NP problem, the verification can in fact be done in polynomial time. i.e. The problem is very difficult to solve, but when you find a solution it is easy to verify. Think of Sudoku, which is of a similar nature.

Problem 3 - Universal Sink

In a directed graph **G**, a **universal sink** is a unique vertex that has incoming edges from all other vertices and no outgoing edges. This means every other vertex in the graph points to the universal sink, but the universal sink points to none. The concept of a universal sink is especially interesting because if one exists in a graph, *it is the only one*.

Example
Consider the graph below:

```
0 → 1
↓ ↙
2 ← 3
```

- Vertex 0 has outgoing edges to vertices 1 and 2 but no incoming edges.
- Vertex 1 has an incoming edge from 0 and an outgoing edge to 2.
- Vertex 2 has an incoming edge from 0, 1 and 3 (no outgoing edges).
- Vertex 3 has an outgoing edge to 2 but no incoming edges.

Vertex 2 is a universal sink because it has incoming edges from all other vertices (0, 1 and 3) and no outgoing edges.

Design and implement an algorithm that tests if a graph ***G***, represented as an **adjacency matrix**, has a universal sink. First write your algorithm in pseudocode, then write it in Java. Your algorithm should run in ***O(n)*** time. For this problem, use an adjacency matrix implementation.

Reflection
- The key insight is that if a universal sink exists, there can be only one such vertex. What does this mean for the algorithm?
- How can you find out if the node is the "source of no edges"?
- After writing the algorithm, think about the significance of using two pointers (i and j) to traverse the adjacency matrix. Why does incrementing i when `G[i][j]` is `1` (and `j` when it's `0`) effectively lead you to a potential universal sink?
- How can you show that your algorithm runs in ***O(n)*** time? Can you think of a worst-case input?

TEACHER POINTERS

1. Start by explaining what a universal sink is in the context of a directed graph. Use the provided example to illustrate the concept. **Uniqueness:** Emphasize that if a universal sink exists, it is unique. This is a critical point for understanding the problem and the algorithm.
2. Discuss how a directed graph can be represented using an **adjacency matrix** and what this representation means. Ensure students can interpret rows and columns in the adjacency matrix in terms of outgoing and incoming edges.
3. **Algorithm Development:** Encourage students to write the algorithm in pseudocode before implementing it in Java. This helps clarify their understanding and logic.
    a. **Finding a Candidate:** Guide students on how to traverse the adjacency matrix to find a potential universal sink. Discuss why incrementing i and j leads to finding a candidate.
    b. **Verifying the Candidate:** After identifying a potential universal sink, explain how to verify it by checking its row and column in the matrix.

   c. Discuss why these specific checks (all zeros in the row, all ones except for the diagonal in the column) are necessary and sufficient to confirm a universal sink.
4. Break down the algorithm's steps to show that each part runs in O(n) time, leading to an overall O(n) time complexity. Worst-Case Scenario: Encourage students to consider and discuss what a worst-case input might look like and how it affects the algorithm's performance.

EXAMPLE SOLUTION

The key insight is that if a universal sink exists, there can be only one such vertex. We can find a candidate vertex that might be a universal sink by traversing the adjacency matrix, and then verify if it indeed is a universal sink.

Algorithm to Find Universal Sink

1. **Find Candidate Vertex:**
   a. Initialize two pointers, `i` and `j`, to `0`. Iterate through the matrix using these pointers. Increment i if `G[i][j]` is 1 (indicating `i` can't be a sink), and increment `j` if `G[i][j]` is 0.
   b. Continue until one of the pointers goes out of bounds. The remaining `i` index is the potential universal sink.
2. **Verify the Candidate Vertex:**
   a. Check the `i`-th row to ensure all entries are `0`.
   b. Check the `i`-th column to ensure all entries are `1`, except `G[i][i]`, which should be `0`.
   c. If both conditions hold, vertex `i` is the universal sink; otherwise, no universal sink exists.

Pseudocode

```
Function isUniversalSink(G):
    n = size of G
    i, j = 0, 0

    While i < n and j < n:
        If G[i][j] is 1:
            i += 1
        Else:
            j += 1

    For k from 0 to n-1:
        If k ≠ i and (G[i][k] is 1 or G[k][i] is 0):
            Return false

    Return true
```

An implementation in java can be found in the “answers” directory.

Complexity Analysis

1. **Traversal to Find Candidate Universal Sink:**
   - The algorithm uses two pointers, `i` and `j`, initialized to 0, and iterates through the adjacency matrix.

    - In each iteration, one of the pointers is incremented based on the value of `G[i][j]`. Specifically, `i` is incremented if `G[i][j]` is 1 (indicating vertex `i` cannot be a sink since it has an outgoing edge), and `j` is incremented if `G[i][j]` is 0.
    - This process continues until one of the pointers goes out of bounds of the matrix. The important observation here is that each pointer can be incremented at most `n` times, where `n` is the number of vertices in the graph. Therefore, this part of the algorithm runs in O(n) time.
2. **Verification of the Candidate Vertex:**
    - After identifying a potential universal sink (`i`), the algorithm verifies whether this vertex is indeed a universal sink.
    - Verification involves two checks:
        - Check if all elements in the `i`-th row are 0 (no outgoing edges from vertex `i`).
        - Check if all elements in the `i`-th column are 1, except for `G[i][i]`, which should be 0 (incoming edges from all other vertices to vertex `i`).
    - Both of these checks require scanning `n` elements (once for the row and once for the column). Hence, each check runs in O(n) time.
3. **Overall Complexity:**
    - The complexity of the algorithm is dominated by these two parts: finding the candidate and verifying it.
    - Since each part runs in O(n) time, the overall time complexity of the algorithm is O(n) + O(n) = O(n).